Towards a Methodology for Modeling with Petri Nets

Christine Choppy and Laure Petrucci
LIPN, UMR CNRS 7030, Institut Galilée - Université Paris XIII
99 Avenue Jean-Baptiste Clément, F-93430 Villetaneuse, FRANCE
email: {Christine.Choppy,Laure.Petrucci}@lipn.univ-paris13.fr

Abstract

Formal specifications remain difficult to write in general, due to both the complexity of the system to be developed, and the use of a formal language. In [4], a method is proposed for specification development, with CASL, the Common Algebraic Specification Language, and CASL-LTL, an extension for dynamic systems specification, as target languages. However, this method could be used with quite a variety of modeling languages, as shown in this paper which is a first attempt to provide systematic guidelines for Petri net specification on the ground of the aforementioned specification method. It is shown how to express in terms of Petri nets the constituent features and the properties exhibited from the first specification approach. A model train specification from [2] is used as a running example.

1 Motivation

While formal specifications are well advocated when a good basis for further development is required, they remain difficult to write in general. Among the problems are the complexity of the system to be developed, and the use of a formal language. So potential helps are needed to start the specification, and then some guidelines to remind some essential features to be described. In [4], a method is proposed for specification development, with CASL[3], the Common Algebraic Specification Language, and CASL-LTL[13], an extension for dynamic systems specification, as target languages. However, this method could be used with quite a variety of target languages.

Petri nets have been successfully used for concurrent systems specification. Among its attractive features, is the combination of a graphic language and an effective formal model that may be used for formal verification. Expressivity of Petri nets is dramatically increased by the use of high-level/coloured Petri nets, and also by the addition of modularity features. Thus, quite sizable examples were specified with Petri nets.

While the use of Petri nets becomes much easier with the availability of high quality environments and tools, to our knowledge, little work was devoted to

a specification methodology for Petri nets. The aim of this work is to provide guidelines for Petri net specification on the grounds of the aforementioned specification method. A train specification [2] is used as a running example.

The structure of the paper is as follows. We first describe the train example in Section 2, then the general specification method [4] is presented in Section 3. The proposed guidelines for Petri net specification are presented in Section 4, together with their application on the train example, and the Petri net specification is given in Section 5.

2 The model train example

Our running example will be the toy railway from [2], in which a step-by-step modeling of the railway by students was described.

The project assigned to students was not only designed as an approach to parallel programming, but also to emphasize the benefits of specification and validation prior to programming. In particular, the students were asked to produce a graphical model, having the same appearance as the physical railway. This was not required for aesthetic reasons but because it greatly helps to understand whether a configuration of the railway is correct or not. This eases a boring and error-prone task of synthesizing a long sequence of transitions. It represents an important benefit for debugging. It also permits to make a direct correspondence between the physical train devices and the Petri net model.

The physical model railway is depicted in Figure 1. It consists of about 15 meters of tracks, divided into 16 sections (blocks B1 to B16) plus 2 sidetracks (ST1 and ST2), connected by four switches and one crossing. The way the trains can pass the switches and the crossing is indicated by the arrows in Figure 1. The traffic on all tracks can go both ways. Although one can notice that switch 1 (and also switch 2) is composed of two elementary ones, it is managed as a single unit, due to the short distance between the two physical components. The railway is connected to a computer via a serial port which allows to read information from sensors and send orders to trains through the tracks or directly to switches. Each section is equipped with one sensor at each end, to detect the entrance or exit of a train. The orders sent to trains can be either stop or go forward/backwards at a given speed.

Hierarchical coloured Petri nets [11] were chosen as a model, due to their tool support for hierarchies, simulation, and occurrence graphs, e.g. DESIGN/CPN [12, 10]. Hierarchies allowed a structured design, where the top-level net reflects the hardware layout. The use of high-level nets permits both capturing several cases by a single transition and representing the parameters of trains and track sections by one place. The use of an ordinary net leads to unreadable intricate models.

The model described in [2] adopts an adaptive routing strategy for the trains

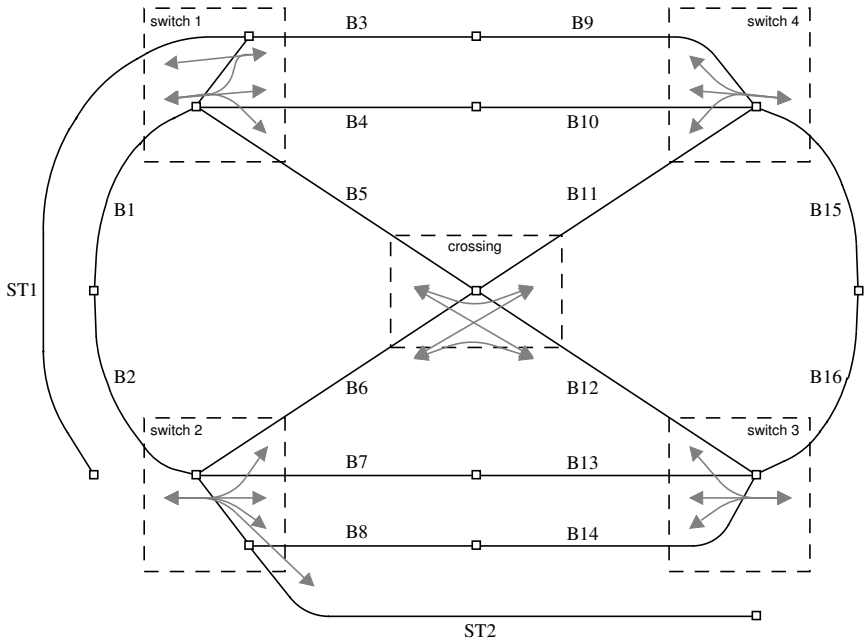

Figure 1: The tracks of the model railway.

to circulate. Hence, the behavior of trains adapts to local conditions. Namely, at each switch, the train's route can be chosen among several tracks and a train may even go back when it cannot continue forward.

Although surprising at the first glance, such behavior of trains offers several complex routing possibilities, demanding the students to design a routing policy so that safety and operational requirements are fulfilled.

3 Specification method principles

The method presented in [4] aims at helping a modeler in designing a "*software item*". It assumes that a *software item* may be either of the following:

- a *simple dynamic system* (a dynamic interacting entity in isolation, e.g., a sequential process) or
- a *structured dynamic system* (a community of mutually interacting entities, simple or also structured), or
- a *data structure* (or data type).

Items are characterized by their *parts* and *constituent features*, that are subsequently specified. For instance, the parts of simple systems are data structures, and their constituent features are states and elementary interactions def-

initions (cf. Section 3.1). The method also involves quite a precise guidance on which properties should be expressed, and in which way.

Among the various specification styles, the *property-oriented* (or *axiomatic*) and *constructive* (or *model-oriented*) ones are mostly used, and here we shall focus on the property-oriented one which is relevant at the beginning of the specification task. In any case, [4] advocates that a visual presentation should be provided to help reading the formal specification, and also that comments should be used, e.g., to accompany formulae.

Property-oriented specification The semantics of *property-oriented specification* is basically defined as follows: "a model belongs to the semantics of a property-oriented specification if and only if all formulae of the specification are valid on it".

The methodological ideas supporting this specification style are:
the item is described at a certain moment in its development by expressing all its "relevant" properties using sentences provided by the formalism (formulae).

For each software item, the *property-oriented* specification technique, is given, by providing the abstract structure of the corresponding specifications together with the related visual presentation and corresponding formal specification.

The target languages are initially CASL[3], the Common Algebraic Specification Language, and CASL-LTL[13], an extension designed for the dynamic systems specification by giving a CASL view to LTL, the Labeled Transition Logic ([1, 9]). LTL, and thus CASL-LTL, is based on the idea that a dynamic system is considered as a *labeled transition system* (shortly *lts*), and that to specify it one has to specify the labels, the states and the transitions of such a system. Recall that an *lts* is a triple $(State, Label, \rightarrow)$, where $\rightarrow \subseteq State \times Label \times State$.

Subsequent work [5, 7, 6] showed that this method could also be used with other target languages, e.g., UML. Although UML is not a formal language, the formally grounded approach used there conveys a quite systematic development for the description, and of course, OCL may be used to describe some of the properties.

In the following, we focus on simple systems items since they are used in the first step when applying our method. Structured systems will be discussed in the conclusion and addressed in further work.

3.1 Simple systems

Here the word *system* denotes a dynamic system of any kind, and so evolving with time, without any assumption about other aspects of its behaviour. Thus it may be a communicating/nondeterministic/sequential/... process, a reactive/parallel/concurrent/distributed/... system, but also an agent or an agents system. A *simple system* is a system without any internal components cooperating together.

Simple systems are seen formally as *labeled transition systems*. The states of an *lts* modeling a simple system represent the relevant intermediate situations in the life of the system, and each transition $s \xrightarrow{l} s'$ represents the *ability* of the system in the state/situation s of evolving to the state/situation s'; the label l contains information on the conditions on the external environment for this ability to become effective, and on the transformation induced on this environment by the execution of the transition, i.e., it fully describes the interaction of the system with the external environment during this transition.

To design effective and simple specification methods, the labels are assumed to have the standard form of a set of *elementary interactions*, where each elementary interaction intuitively corresponds to an elementary (that is, not further decomposable) exchange with the external environment. It is also assumed that the elementary interactions are of different types, and that each type is characterized by a name and by some arguments (elements of some data structures). Thus, *elementary interaction types* (just *elementary interactions* from now on) are constituent features of the simple systems.

The form of the states (which are the intermediate situations during the system's life) is also a characterizing feature of simple systems, therefore *state constituent features* are needed. However, they are technically different for the property-oriented and the constructive case.

Finally, to define the constituent features of a simple system, values of various *data structures* are used; they are the "parts" of the simple systems.

3.2 Simple systems property-oriented specifications

The property-oriented specification method for simple systems requires to first find the parts and constituent features, and then to express the properties. In order to keep the specification level abstract, the states are not completely described, but only a list of what should be observed is given, and thus the state features will correspond to elementary observations on the states (*state observers*). A state observer is characterized by a name, some arguments (elements of some *data structures*), and by the observed value (element of some data structure). Figure 2 shows the structure (by means of a UML class diagram[1]) of a property-oriented specification of a simple system, and Figure 3 shows how to visually depict its parts (DATA_1, ..., DATA_j) and the constituent features.

3.3 Simple systems properties

All the properties about a simple system correspond to properties on the *lts* modeling it, and thus on its labels, states and transitions. These properties may express which are the admissible sets of *elementary interactions* building a label, and link the source state, the label and the target state of a transition.

[1] To shortly explain the UML notation, the diamond connects the "Simple system specification" with its constituents, the * indicates the multiplicity (as in regular expressions), and labels on the lines provide a "role" name for each part.

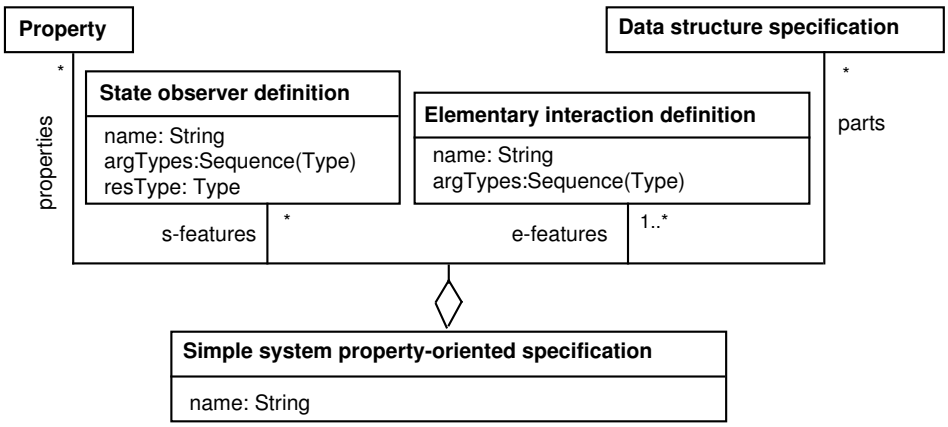

Figure 2: Simple System Property-Oriented Specification.

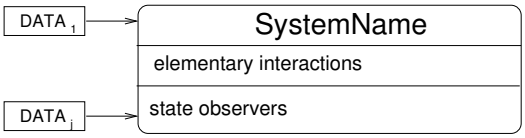

Figure 3: Visual presentation of a simple system: parts and constituent features.

The properties may also provide some information on the values observed by the various state observers on a state.

More precisely, *label properties* express when, under some condition, two different elementary interactions are incompatible, i.e., no label may contain both (cf. incompat1 and incompat2 in Figure 4). *State properties* describe conditions the values returned by the state observers should satisfy for any state (cf. value1 and value2 in Figure 4). State formulae may also include special atoms, expressing properties on the *paths* (concatenated sequences of transitions) leaving/reaching the state, that is on the future/past behaviour of the system from this state. *Transition properties* are conditions on the state observers applied to the source and target states of the transition.

Guidelines for properties follow a general tableau method which gives provision for "property cells" with respect to the system constituent features. Since the constituent features of simple systems are of two kinds, elementary interactions and state observers, five kinds of "property cells" are considered:

- properties on an elementary interaction,
- properties on a state observer,
- relationship between two elementary interactions,
- relationship between two state observers,

- relationship between an elementary interaction and a state observer.

Schemas for these five property cells are described in Figure 4, with, for each cell, the list of possible properties.

In Figures 5 and 6 the details of two schemas are given, providing, for each property, its name, an informal comment, and its formal expression in a visual presentation associated with CASL-LTL. There, *arg* stands for generic expressions of the correct types, possibly with free variables, and *cond*(*exprs*) for a generic condition where the free variables of *exprs* may appear.

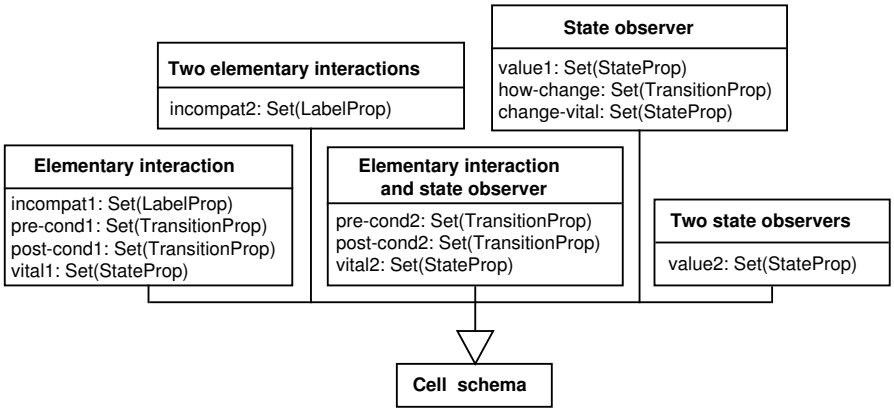

Figure 4: Simple System Cell schemas

4 Applying the specification approach to the train example

The general ideas [4] described in Section 3 were initially introduced to help designing an algebraic specification. We will show here that these principles can also be applied to coloured Petri nets, through the model train example.

4.1 Parts and constituent features

In order to apply the general ideas given in Section 3, we first need to choose what kind of software item our system is. When dealing with systems like the model train example, we may consider, in a first approach, that a single entity is involved (the railway), and therefore that it has to be specified as a *simple dynamic system*. This will lead to a general high-level design of the system, not getting into the details of trains changing sections policies.

According to Figure 2 need to find the system (sub)parts and constituent features. In both cases, the parts are the data structures required in the system.

pre-cond1 (transition property) If the source state of the transition satisfies some condition then the label of a transition contains some instantiation of *ei*.

if *cond*(*arg*) **then** *ei*(*arg*) **happens**
where some source state observers must appear in *cond*(*arg*) and the target state observers cannot appear in *cond*(*arg*).

post-cond1 (transition property) If the label of a transition contains some instantiation of *ei*, then the target state of the transition must satisfy some condition. The condition on the target state may require also the source state to be expressed.

if *ei*(*arg*) **happens then** *cond*(*arg*)
where some target state observers must appear in *cond*(*arg*) and the source state observers may appear in *cond*(*arg*).

incompat1 (label property) Two instantiations of *ei* are incompatible (i.e., no label may contain both) if their arguments satisfy some conditions.

ei(arg_1) **incompatible with** *ei*(arg_2) **if** *cond*(arg_1,arg_2)

vital1 (state property) If a state satisfies some condition, then any path (sequence of transitions) starting from it will eventually contain a transition whose label contains *ei*. Note that in these properties **in any case** may be replaced by **in one case** and **eventually** by **next**.

if *cond*(*arg*) **then in any case eventually** *ei*(*arg*) **happens**

Figure 5: Elementary interaction (*ei*) cell schema

value1 (state property) The results of the observation made by *so* on a state must satisfy some conditions.

cond, where *so* must appear in *cond*.

how-change (transition property) If the observed value changes during the occurrence of a transition, and some elementary interactions belong to the transition label, then some condition on source and target states, old and new values holds (new values are denoted with a $'$).

if $so(arg) = v_1$ **and** $ei_1, \ldots, ei_n$ **happened**
then $so'(arg) = v_2$ **and** $v_1 \neq v_2$ **and** $cond(v_1,v_2,arg)$

change-vital (state property) If a state satisfies some condition, then the observed value will change in the future. Note that in these properties **in any case** may be replaced by **in one case** and **eventually** by **next**.

if $cond(v_1,v_2,arg)$ **and** $so(arg) = v_1$ **and** $v_1 \neq v_2$ **then**
in any case eventually $so(arg) = v_2$

Figure 6: State observer (*so*) cell schema

The constituent features are the elementary interactions and the state description features (observers or constructors). At this first stage of model design, the property-oriented approach is often more relevant, thus we need state observers to start with.

The physical system is made of track sections, switches between track sections, and trains. Thus, *state observers* should provide information on the layout of tracks, i.e. which track sections are contiguous, which ones are connected by switches, whether a train is present on a track, and, when this is the case, in which direction it is traveling (this may be expressed in various ways, e.g. here, clockwise or anticlockwise).

The *elementary interactions* (that are associated with a state change of the system) are a train track section change, moving either directly between contiguous sections or between sections connected by a switch. It is admitted that the position of a track section is fixed (sic!), and that the potential connections that can be established by a given switch are also fixed, and this will be reflected in the state observers properties.

The required *data structures* are obtained through the data types used by the state observers and the elementary interactions. Quite obviously, some data type is needed to refer to track sections, and to switches. Since they are named in Figure 1 (i.e., the possible values are known and in a quite limited number), so-called enumerated types are adequate. The same principle is used

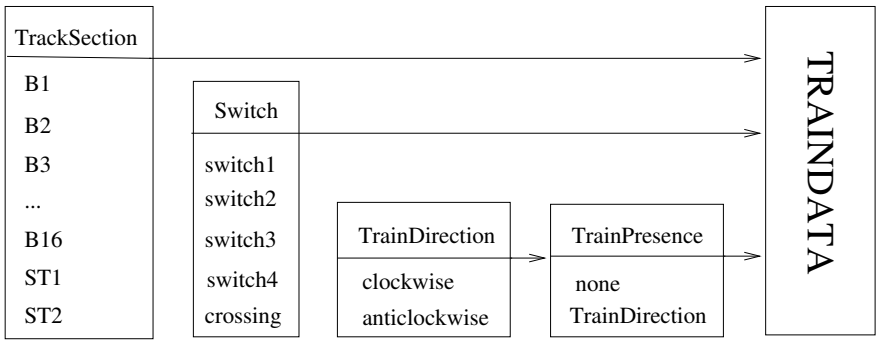

Figure 7: The data structures.

for train directions, as shown in Figure 7. We present here also the corresponding CASL specification for these data where the type name is simply followed by the enumeration of its possible values (which are constants of this type). The **free** construct insures that no property relates (e.g., equates) these values, so that they are all different. The **sort** construct is used here to express that any element of the type $TrainDirection$ is also of the type $TrainPresence$.

spec TRAINDATA =
  free type
    $TrackSection ::= B1 \mid B2 \mid B3 \ldots \mid B16 \mid ST1 \mid ST2$
  free type
    $Switch ::= switch1 \mid switch2 \mid switch3 \mid switch4 \mid crossing$
  free type
    $TrainDirection ::= clockwise \mid anticlockwise$
  free type
    $TrainPresence ::= none \mid$ **sort** $(TrainDirection)$
  end

These data will be reflected either in the names of states and transitions of the Petri net, or as colours of tokens.

The *state observers* are chosen so as to provide enough information on the state of the modeled system. For our example, observers are needed to describe the track sections layout, as well as the presence of a train with its travel direction (Figure 8). The *connected* predicate is used to express when two track sections are directly connected, and in which train direction. The *switched* predicate is used to express when two lists of track sections are connected through a switch, and in which train direction. These observers (*connected* and *switched*) are fixed once the railway topology is fixed. This is not the case for *train_present* which reflects a situation that evolves with time, and that depends on the initial state as well as the history of elementary interactions leading to the current state. The state and history type specifications are given below.

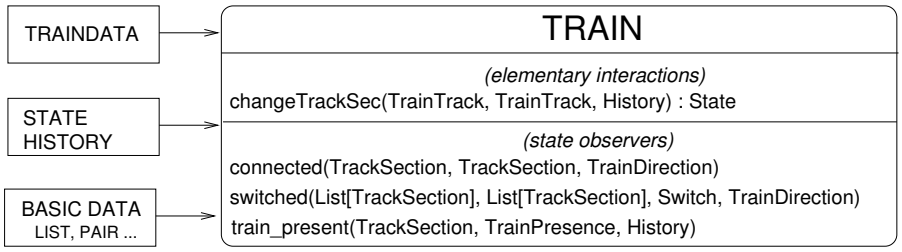

Figure 8: The train elementary interactions and state observers.

spec STATE =
  sort *State*;
  op *initial* : *State*; %% There is an initial state
    %% which will be further described in the Section 5
  end

spec HISTORY = STATE **then**
  type $History ::= initial \mid \_\_.\_\_(History; State)$;
  op $last : History \rightarrow State$;
  vars $h : History; s : State$;
  axioms
    $last(initial) = initial$;
    $last(h.s) = s$;
  end

The *elementary interactions* express the fact that a train changes track section (Figure 8).

4.2 Properties

Once the parts and constituent features of the system are specified, its properties should be expressed. Following the method described in Section 3.3, the property cells of Figure 4 should be filled. Since there is only one elementary interaction, the relationship between two elementary interactions is skipped.

Properties on a state observer

value1 (state property) Here we express the results of observations. The properties on *connected* and *switched* do not change and express the railway topology. The properties on *train_present* vary with the state.

$connected(B1, B2, anticlockwise)$
$connected(B2, B1, clockwise)$
...

$switched((ST1, B1), B3), switch1, clockwise)$
$switched((B1), (B3, B4, B5), clockwise)$
$switched((B3), (B1, ST1), anticlockwise)$
$switched((B3, B4, B5), (B1), switch1, anticlockwise)$
...
$train\_present(B1, none, initial)$ ...

how-change (transition property) As mentioned above, this concerns only *train_present* which varies when a track section change $changeTrackSec$ occurs.

if $train\_present(TS_i, TP_i, h) \wedge train\_present(TS_j, none, h)$
$\wedge\ changeTrackSec(< TS_i, TP_i >, < TS_j, none >, h)$ **happened**
then $(TP_i \neq none) \wedge train\_present(TS'_i, none, h') \wedge$
$train\_present(TS'_j, TP_i, h')$
where h' denotes $h.changeTrackSec(< TS_i, TP_i >, < TS_j, none >, h)$

change-vital (state property) This property is not relevant here.

Properties on the elementary interaction $changeTrackSec$

pre-cond1 (transition property) A track section change is defined when the two track sections are connected or "switched", when there is a train traveling in the (connection or switch) direction in the first track section, and no train in the second one.

if $(connected(TS_i, TS_j, TP_i) \vee$
$\exists sw : Switch\ s.t.\ swichted((\ldots, TS_i, \ldots), (\ldots, TS_j, \ldots), sw, TP_i))$
$\wedge\ (TP_j = none)$
then $changeTrackSec(< TP_i, TS_i >, < TP_j, TS_j >, h)$ **happens**

post-cond1 (transition property) After a track section occurred, the train is in the target track section.

if $changeTrackSec(< TP_i, TS_i >, < TP_j, TS_j >, h)$ **happens then** $(TP'_j = TP_i)$

incompat1 (label property) This property should express when simultaneous train track section changes should not occur. Since the information on the direction of the train is included in the interaction, the only case is that, at a given switch, a train cannot take simultaneously several directions.

$changeTrackSec(< TP_i, TS_i >, < TP_j, TS_j >, h)$ **incompatible with**
$changeTrackSec(< TP_i, TS_i >, < TP_k, TS_k >, h)$
if $\exists sw : Switch\ s.t.\ swichted((\ldots, TS_i, \ldots), (\ldots, TS_j, \ldots, TS_k, \ldots), sw, TP_i)$
$\wedge (T_j \neq T_k)$

vital1 (state property) There is no property here since it is not relevant here to express that a track section change will eventually happen.

There are no properties between the state observers, and the properties expressing the relationship between the elementary interaction and the *train_present* state observer are redundant with those already expressed.

In the methodology introduced here, some properties can be specified, which are not part of the Petri net model per se. For example, the modeler could specify a state property (see Figure 6) expressing that, unless otherwise imposed by the initial state, there is always a single token in each place representing a track section (which is inferred by the ***pre-cond1*** of $changeTrackSec$ above).

Even though this property seems extremely simple, it is important to guide the modeler into explicitly writing down the expected properties from the system, based on the current status of the model being designed.

In later phases of system development, a simple system can evolve by refining its constituents, or by composing it with other systems. Stating expected properties is then crucial to have better insight. These properties could be verified by a model-checking tool, in order to check consistency of the model w.r.t. the intended behaviour.

5 From the specification to the coloured Petri net

The *state observers* are reflected in the Petri net in different ways. The *fixed part*, that is here the way track sections are connected, together with the potential switch connections, may be reflected by the Petri net layout, as suggested in the pedagogical project of [2], thus it will be observable on the grounds that it will be possible for a train to move from one track section to another (connected) one. More precisely, the Petri net places reflect the different track sections, and places that model adjacent track sections are connected with transitions associated with a train changing track section. Following [2], it is suggested that places and transitions are displayed so as to reflect the physical model train track and switches display.

Quite obviously then, *elementary interactions* reflecting a train changing track section are specified by the corresponding transitions.

The presence of a train together with its direction (none, clockwise or anticlockwise) comes here as a *colour* for the track section places.

The *pre-conditions* and *post-conditions* properties of the elementary interactions (see Figure 5) induce the arcs between places and transitions.

Hence, we obtain a model which is similar to the prime page of [2] presented in Figure 9.

The prime page represents the whole railway, without any consideration of the policy used to move from one section to the next. This policy is described in

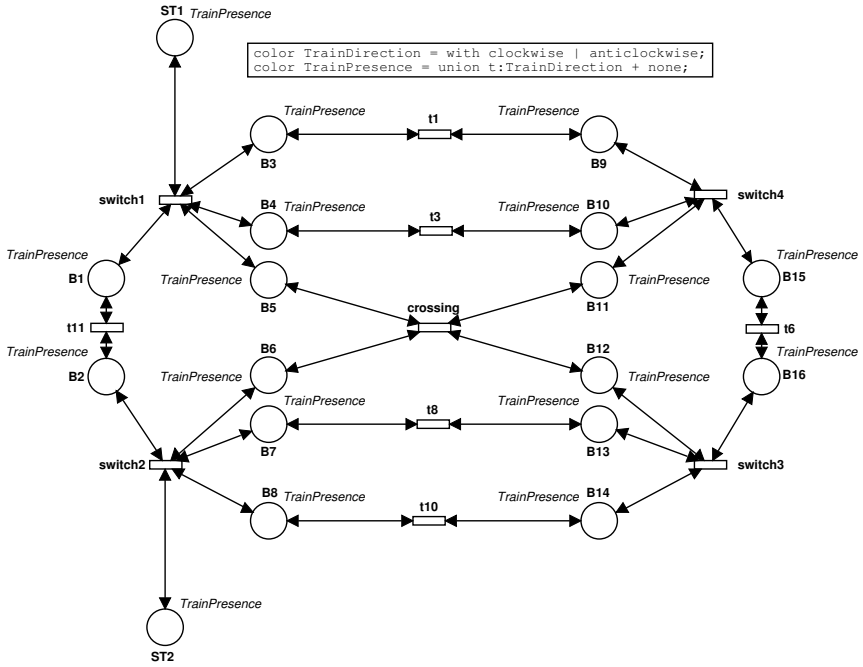

Figure 9: The prime page of the model railway hierarchical coloured Petri net.

sub-pages, corresponding to the different switches and moves between adjacent sections. A single look at this prime page shows the current state, i.e. where the different trains are located. The similarity between the physical railway model (Figure 1) and the prime page (Figure 9) is easily observed. The places represent the sections (they have the same names in both figures), while the transitions indicate the possible moves.

The colors (data types) of tokens within places are defined in the *global declaration node* (boxed text at the top of Figure 9). First, the direction of a train, `TrainDirection`, can be either `clockwise` or `anticlockwise`. Each place represents a railway section with the corresponding name and thus always contains one token of color `TrainPresence`, with a value characterizing the state of the section, that is either a train is in the section, or the section is empty. This is expressed with the union type:

```
color TrainPresence = union t:TrainDirection + none;
```

All the transitions are substitution transitions, i.e. their behaviour is explicited on the associated subpage. They precisely describe the policy used to change sections. In this paper, we will not get into the details of these policies.

Now, the initial situation chosen for the railway is that there are trains traveling in the clockwise direction on track sections B9 and B10, and trains

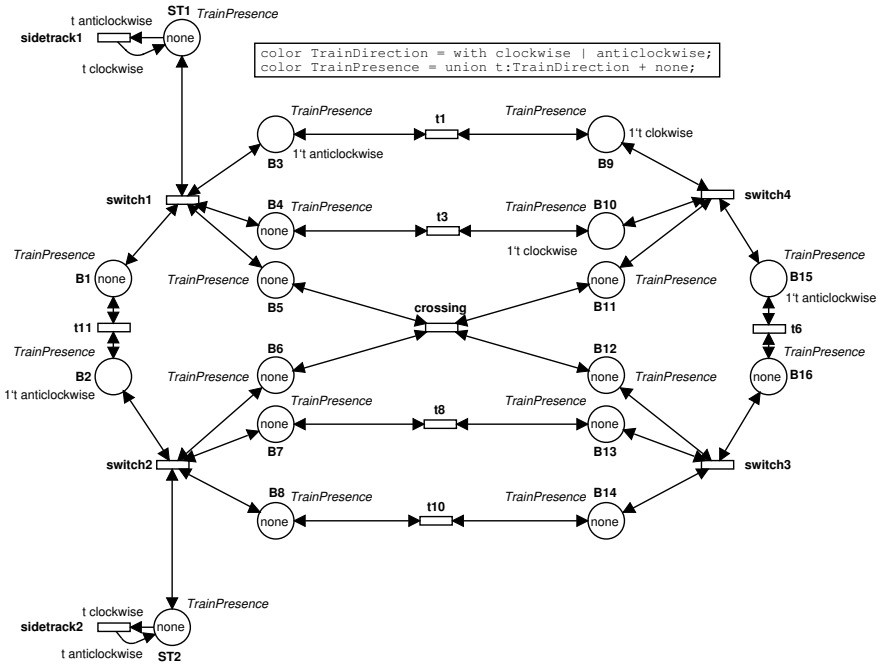

Figure 10: The prime page with the initial marking.

traveling in the anticlockwise direction on track sections B2, B3 and B15. This is reflected by the following properties:

$train\_present(B9, clockwise, initial)$;
$train\_present(B10, clockwise, initial)$;
$train\_present(B2, anticlockwise, initial)$;
$train\_present(B3, anticlockwise, initial)$;
$train\_present(B15, anticlockwise, initial)$;

which are represented by an initial marking in the Petri net of figure 10.

6 Conclusion and perspectives

In this paper, we provided guidelines for specifying "simple systems" using Petri nets. These guidelines are derived from a method developed in [4] for an algebraic specification language and an extension for dynamic systems specification. In particular, elements provided for specifying "simple systems" (consisting of a single dynamic entity) were studied for their expression with Petri nets. The (sub)parts are data structures (that can be used e.g. for the Petri net colors), and we refer to the method in [4] for their specification. The state description features may be reflected either in the Petri net layout (i.e., the way places and transitions are connected), or in the information conveyed in the places. The

elementary interactions are reflected in the transitions firings. The properties for the state descriptors and the elementary interactions may be checked against the Petri net properties or behaviour.

This first experiment with a model train seems quite promising in the direction of providing a more extensive method for Petri net specification. It shows that the methodology envisioned applies to a large panel of specification languages, which are in essence quite different.

This work should be pursued by extending the "structured systems" part of the approach described in [4]. Our approach should then be extended so as to include the communication mechanisms between modules provided by Petri nets (e.g. hierarchical CPNs [11], modular Petri nets [8]). This should also include property verification, i.e. if a general property is to be satisfied, it would be nice to know at which level of the specification process a formal analysis should (in)validate it. Applying this methodology to design step-by-step a complex case study is another important issue.

References

[1] E. Astesiano and G. Reggio. Labelled Transition Logic: An Outline. *Acta Informatica*, 37(11-12):831–879, 2001.

[2] G. Berthelot and L. Petrucci. Specification and validation of a concurrent system: An educational project. *Journal of Software Tools for Technology Transfer*, 3(4):372–381, 2001.

[3] M. Bidoit and P.D. Mosses. *CASL User Manual, Introduction to Using the Common Algebraic Specification Language*. Lecture Notes in Computer Science 2900. Springer-Verlag, 2004.

[4] C. Choppy and G. Reggio. Towards a Formally Grounded Software Development Method. Technical Report DISI–TR–03–35, DISI, Università di Genova, Italy, 2003. Available at `ftp://ftp.disi.unige.it/person/ReggioG/ChoppyReggio03a.pdf`.

[5] C. Choppy and G. Reggio. Improving use case based requirements using formally grounded specifications. In *Fundamental Approaches to Software Engineering*, LNCS 2984, pages 244–260. Springer Verlag, 2004.

[6] C. Choppy and G. Reggio. A uml-based method for the commanded behaviour frame. In K. Cox, J.G. Hall, and L. Rapanotti, editors, *Proc. of the 1st International Workshop on Advances and Applications of Problem Frames (IWAAPF 2004)*, pages 27–34. An ICSE 2004 workshop, IEEE, 2004.

[7] C. Choppy and G. Reggio. Using uml for problem frame oriented software development. In Walter Dosch and Narayan Debnath, editors, *Proc of the*

ISCA 13th Int. Conf. on Intelligent and Adaptative Systems and Software Engineering (IASSE-2004), pages 239–244. The International Society for Computers and Their Applications (ISCA), 2004.

[8] S. Christensen and L. Petrucci. Modular analysis of Petri nets. *The Computer Journal*, 43(3):224–242, 2000.

[9] G. Costa and G. Reggio. Specification of Abstract Dynamic Data Types: A Temporal Logic Approach. *T.C.S.*, 173(2):513–554, 1997.

[10] DESIGN/CPN *online*. `http://www.daimi.au.dk/designCPN`.

[11] K. Jensen. *Coloured Petri Nets: Basic concepts, analysis methods and practical use. Volume 1: basic concepts*. Monographs in Theoretical Computer Science. Springer, 1992.

[12] META Software and Aarhus University. *Design/CPN 3.0*, 1996. Also available as: `http://www.daimi.au.dk/designCPN`.

[13] G. Reggio, E. Astesiano, and C. Choppy. CASL-LTL : A CASL Extension for Dynamic Reactive Systems Version 1.0– Summary. Technical Report DISI-TR-03-36, DISI – Università di Genova, Italy, 2003. Available at `ftp://ftp.disi.unige.it/ person/ReggioG/ReggioEtAll03b.ps` and `ftp://ftp.disi.unige.it/ person/ReggioG/ReggioEtAll03b.pdf`.